Comparison of Ray Tracing Acceleration Structures
kD Trees vs. Tetrahedralizations

Bradford J. Loos
University of Utah

1 Introduction

This paper investigates the build, space, and query times for two different ray tracing acceleration structures, kD trees, and tetrahedralizations. It then compares the two and finds that while kD trees are better in basically all metrics, tetrahedralizations may be useful enough to consider for use due to other factors besides speed and space.

2 Ray Tracing

Ray tracing is the act of tracing rays through a space and trying to determine which objects, if any, are hit by an incoming ray. In computational geometry this problem is related to the line stabbing problem – given a set of objects what is the maximum number of objects that can be hit by a single straight line through the scene.

In this paper, we are talking about ray tracing as it relates to graphics. Given a scene, an eye point, and an image plane ray tracing is defined as follows:

- Discretize the image plane into pixels
- Shoot a ray from the eye point (origin) through each pixel (direction)
- Intersect each ray with all the objects in the scene
- Calculate the closest intersection
- Cast more rays from the intersection point for shading calculations

2.1 Acceleration Structures

Due to the fact that intersecting each ray with each triangle in the scene requires too much time, different acceleration structures are used. In the past many different structures have been used including grids, kD trees, and bounding volume hierarchies (BVH). Also, these structures can be used in a hierarchical manner. Exactly which structure is considered the most beneficial changes from year to year. This paper will be talking about kD trees and a new structure (for ray tracing) called a constrained tetrahedralization.

3 kD Trees

A kD tree is a partition of space in three dimensions. Unlike the usual kD trees, for ray tracing a kD tree contains triangles instead of points and may or may not change the dimension of the split plane on each sub-division. Also instead of simply splitting the objects at their median (as in normal kD trees) we use a heuristic called the surface area heuristic. At each level the split is based on the surface area of the split volume versus the surface area of the original parent volume. Split planes are tried at the edges of each triangle (of which there may be 6) and counting how many triangles fall into each sub-child.

function recBuild(triangles T, voxel V)
    if Terminate(T, V) then
        return new leaf node(T)
    end if
    p = FindPlane(T, V)
    (V_L, V_R) = Split V with p
    T_L = \{ t | t \cap V_L \neq \emptyset \}
    T_R = \{ t | t \cap V_R \neq \emptyset \}
    return new node (recBuild(T_L, V_L), recBuild(T_R, V_R))
end function

Figure 1: Pseudocode function to recursively build a kD tree.

3.1 Building a kD Tree

The basic algorithm used to recursively create a kD tree is fairly simple (see Figure 1) but requires us to define a termination condition and a method to find the split planes.

Termination Criteria The surface area heuristic gives us a simple termination condition for the algorithm. If the traversal cost of the children using the minimum cost splitting plane would be greater than the cost of simply intersecting the ray with all the children then the algorithm should terminate and not split.

\[ \text{Terminate}(V, T) = \begin{cases} \text{true} & : \min_{p \in |V|} C(p) > K_T |T| \\ \text{false} & : \text{otherwise} \end{cases} \]

Find Plane There are many ways to find the split plane for this algorithm. If we weren’t using the surface area heuristic we could simply select the median in each dimension and split that way. This would run in \( O(n \log n) \) but would give undesirable query times (in practice). Instead we will try and find a good splitting plane as shown in the code in Figure 1.

If we want to use the surface area heuristic we need to calculate the cost of creating the tree in a more complex manner. The cost of intersecting an internal node given a plane \( p \) can be derived as follows [Wald et al. 2006]:

\[ C_V(p) = K_T + P_{|V_L|} C(V_L) + P_{|V_R|} C(V_R) \]

Where the probability of hitting each child \( P_{|V|} \) is based on the ratio of it’s surface area to the surface area of it’s parent. To find the best tree we need to optimize this value over the whole tree which results in this function [Wald et al. 2006]:

\[ C(T) = \sum_{n \in \text{nodes}} \frac{SA(V_n)}{SA(V_S)} K_T + \sum_{l \in \text{leaves}} \frac{SA(V_l)}{SA(V_S)} K_l \]

Unfortunately Wald tells us “The number of possible trees, however, rapidly grows with scene size, and finding the globally optimal tree today is considered infeasible except for trivial scenes” [Wald et al. 2006]. Fortunately if we use the assumption that each node’s children will be leaf nodes we can simplify the equation to the cost of a node traversal \( K_T \) plus the cost of traversing all the left child’s triangles \( |T_L| K_l \) times it’s probability \( P(|V_L|) \).

COMP
\[ C_v(p) \approx K_T + P_{VT}(V_L) |T_L| K_T + P_{VT}(V_R) |T_R| K_T \]
\[ = K_T + K_T \left( \frac{SA(V_L)}{SA(V)} |T_L| + \frac{SA(V_R)}{SA(V)} |T_R| \right) \]

The naïve method of find the split plane is simply to check each perfect split \(^1\), of which there are \(O(n)\), as the split plane. Since there are \(n\) triangles that need to be classified for each perfect split the naïve method will take \(O(n^2)\) time.

Wald [Wald et al. 2006] gives another build method that uses the surface area heuristic but still runs in \(O(n \log n)\) time. The basic idea is to use a sweep line algorithm with an event list containing the split planes as events. The first problem you run into is that you would still need to order these by dimension at each level. To avoid this, each event in the original list contains not only the geometric information needed, but also the dimension of the split plane. This way, all dimensions can be contained in a single list avoiding a sorting at each level of the recursion.

The sweep line algorithm keeps 3 lists, those items that are left of the split plane, those items that are on the split plane, and those items that are right of the split plane. Thanks to the original sorting we have an ordered list of those items that are wholly left of the split plane and an ordered list of those items totally to the right. The items that straddle the split plane are also ordered, and so can be integrated into the left and right lists in an ordered manner via a merge operation in \(O(n)\). This allows \(O(n)\) work at each level of the tree resulting in an entire run time of \(O(n \log n)\).

### 3.2 Space Bounds of a KD Tree

The KD tree is simply a binary tree where each internal node takes constant space. With \(n\) leaf nodes this structure would use \(O(n)\) space for the entire structure. Unfortunately due to the fact that the triangles of the model may be split at each level there could be many more than \(n\) leaves which would cause this structure to require more than \(O(n)\) space.

### 3.3 Querying a KD Tree

The process of querying the KD tree is simply:

- Does the ray hit the AABB of the parent
- Based on the direction of the ray select which children to check (None/L/R/LR)
- Recursively check the chosen children

Due to the fact that most of the items are based on the structure of the tree which can change with each model that is used it is nearly impossible to get a valid big-Oh notation for the run time of this query. Although it is relatively close to:

\[ C_{tr} \sum_{i=1}^{N_i} \frac{SA(i)}{SA(root)} + C_{ll} \sum_{i=1}^{N_i} \frac{SA(l)}{SA(root)} + C_{ll} \sum_{i=1}^{N_i} \frac{SA(l)N(l)}{SA(root)} \]

Where the values are defined as follows:

- \(C_{tr}\) Cost of Internal Node Traversal
- \(N_i\) Number of Internal Nodes
- \(C_{ll}\) Cost of Leaf Node Traversal
- \(N_j\) Number of Leaves
- \(C_{ll}\) Cost of Triangle Intersection
- \(N(l)\) Number of Triangles in a Leaf

### 4 Tetrahedralizations

Tetrahedralization is actually just another name for a Delaunay triangulation in 3 dimensions. What is used in the ray tracing paper is actually constrained tetrahedralizations and not true delaunay triangulations. A constrained tetrahedralization allows points to invalidate the usual circumcircle property required of Delaunay triangulations as long as those points are hidden from the selected points by a constrained edge or face.

Unfortunately, not all polyhedra can necessarily be tetrahedralized and figuring out if any given polyhedra is tetrahedralizeable is an NP-Complete problem [Shewchuk 1998a]. Fortunately, every polyhedra can be tetrahedralized if you allow the addition of addition points called Steiner points.

To create our constrained tetrahedralization we change the model we care about into a piecewise linear complex (PLC). As almost all models are made up of triangles and therefore can easily be converted into a PLC.

#### 4.1 Building a Constrained Tetrahedralization

While there is a sweep line algorithm [Shewchuk 2000] that runs in \(O(n\log n)\) time. The basic idea is to use a sweep line algorithm with an event list containing the split planes as events. The first problem you run into is that you would still need to order these by dimension at each level. To avoid this, each event in the original list contains not only the geometric information needed, but also the dimension of the split plane. This way, all dimensions can be contained in a single list avoiding a sorting at each level of the recursion.

The sweep line algorithm keeps 3 lists, those items that are left of the split plane, those items that are on the split plane, and those items that are right of the split plane. Thanks to the original sorting we have an ordered list of those items that are wholly left of the split plane and an ordered list of those items totally to the right. The items that straddle the split plane are also ordered, and so can be integrated into the left and right lists in an ordered manner via a merge operation in \(O(n)\). This allows \(O(n)\) work at each level of the tree resulting in an entire run time of \(O(n \log n)\).

<table>
<thead>
<tr>
<th>Steps</th>
<th>Algorithms</th>
<th>Worst Case</th>
<th>General Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Delaunay Triangulation</td>
<td>(O(n^2))</td>
<td>(O(n \log n))</td>
</tr>
<tr>
<td>(2)</td>
<td>Segment Recovery</td>
<td>(O(n \log n))</td>
<td>(O(n \log n))</td>
</tr>
<tr>
<td>(3)</td>
<td>Vertex Perturbation</td>
<td>(O(n \log n))</td>
<td>(O(n \log n))</td>
</tr>
<tr>
<td>(4)</td>
<td>Facet Recovery</td>
<td>(O(n \log n))</td>
<td>(O(n \log n))</td>
</tr>
</tbody>
</table>

### Segment Recovery

After doing a normal Delaunay triangulation you need to recover the segments in the PLC. To do this you create a list of edges that are in the PLC but not the triangulation. Take one of these lines and put it in the triangulation. Unfortunately this may invalidate your triangulation. So, for each point in the original triangulation that impinges on the circumcircle of the new line needs to be projected onto the new line. This protects the line so that no points in the triangulation impinge on the circumcircle. Both the triangulation and the PLC need to be updated to contain the set of segments instead of just the original edge that was originally placed into the triangulation. This process continues until all the edges in the PLC are in the Delaunay triangulation.

### Vertex Perturbation

After creating all these new segments we may have some degenerate vertices in the mesh. Some of these can be fixed by simply perturbing them. If the point lies on a single segment, the point can be moved along the segment without affecting the PLC. The same idea can be used if the point is found on a face of the PLC. But, some points may not be able to be moved without changing the PLC itself. For this we need to insert a new point in the middle of a face of the PLC and retriangulate that portion of the model. This assures that the PLC is valid and free of degeneracies.

### Facet Recovery

Now that we have all the edges in the graph we need to recreate the faces. To do this we use a cavity retetrahedralization technique [Si and Gärtner 2005]. Basically you put all the faces that need to be recovered into a queue and start deleting edges around those faces. Once you have an empty region around the desired face, the mesh is split into two pieces and the face is inserted. Each side of the cavity is retetrahedralized and then both pieces are rejoined at the selected face.
4.2 Building a Quality Tetrahedralization

Some of the tetrahedralizations created in this fashion have many long and skinny tetrahedra. To fix this problem (to improve the query time) we create what is known as a quality tetrahedralization [Shewchuk 2001]. This makes sure that the circumradius-to-shortest edge ratio of the tetrahedra are minimized creating better tetrahedra for the algorithm to work with.

4.3 Space Bounds of a Tetrahedralization

Since a tetrahedralization is simply a Delaunay triangulation in 3 dimensions we know that the algorithm will create at most \( O(n^{d/2}) \) simplices which reduces to \( O(n^2) \) in this case. Although we must not forget that the number of additional points that may need to be added is \( \Omega(n^2) \) [Si 2007] but this only changes the constant and not the actual big-Oh notation.

4.4 Querying a Tetrahedralization

According to [Shewchuk and D-simplices 2000] it seems that the line stabbing algorithm against tetrahedra created via \( n \) triangles is \( \theta(n^2) \). This can be seen if you create two rows of points and create tetrahedra between them. If a line is stabbed down the row of tetrahedra it will hit each and every one of them, creating \( n^2 \) intersections. But in [Lagae and Dutré 2008], they say that in practice “…the number of tetrahedra traversed by a ray is relatively small.” So theoretically trying to intersect a set of tetrahedra based on a random set of points will be quite slow, but in practice on normal ray tracing scenes the results are not nearly as bad.

5 Comparison

In comparing the two structures we really need to make sure we know in what field the comparison matters. In computational geometry we can see that the comparison highly favors a kD tree\(^2\).

<table>
<thead>
<tr>
<th></th>
<th>kD Tree</th>
<th>Constrained Tets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td>( O(n \log n) )</td>
<td>( O(n^4 \log n) )</td>
</tr>
<tr>
<td>Space</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Query</td>
<td>( O(n^{2}) )</td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>

But if we look at figure 2 we can see that while the number of constrained tetrahedralizations are an order of magnitude slower the actual run times are acceptable enough to be used especially considering that while there has been 20 years of research into speeding up kD trees, the idea of tetrahedral acceleration structures is quite new.

6 Why would you use this

So, now that we have seen that a tetrahedralization is much slower using big-Oh notation and also slower than a kD Tree in practice, why would we want to use such a structure as our acceleration method?

**Dynamic Geometry** Because the acceleration structure is based on the points in the scene deformations of the model do not neccessarily constitute a re-calculation of the acceleration structure. “If the deformation field is \( C^1 \) continous and divergence-free then no local or global self intersections can occur” [Lagae and Dutré 2008]. Although it would probably work best in a rigid hierarchy since even a simple global rotation would cause the tetrahedralizations to intersect.

**Global Illumination** Other data could also be associated with points or faces in the tetrahedralization. Due to this fact we could store information from global illumination processes such as photon maps, or irradiance caches inside our original acceleration structure avoiding expensive lookups into other spatial data structures.

**GPU Implementations** Since the tetrahedralization queries are not recursive there is no need for a stack. This fact lends itself to implementation on other hardware architectures such as GPUs (which have no stack).

So, even though the structure itself is slower it may lead to new directions in research that may not have been visited if only the traditional acceleration structures were considered.

References


Figure 2: Results for kD Tree, Constrained Tetrahedralization, and Quality Constrained Tetrahedralization from Lagae

<table>
<thead>
<tr>
<th></th>
<th>Neptune</th>
<th>Forest</th>
<th>Chair</th>
<th>Large Chair</th>
<th>Small Chair</th>
<th>Armadillo</th>
</tr>
</thead>
<tbody>
<tr>
<td># triangles</td>
<td>2.64M</td>
<td>17.52k</td>
<td>408.41k</td>
<td>425.92k</td>
<td>425.92k</td>
<td>345.95k</td>
</tr>
<tr>
<td><strong>Constrained Tetrahedralization</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># tetrahedra</td>
<td>8.53M</td>
<td>63.08k</td>
<td>1.43M</td>
<td>1.48M</td>
<td>1.46M</td>
<td>1.21M</td>
</tr>
<tr>
<td># faces</td>
<td>17.07M</td>
<td>126.17k</td>
<td>2.86M</td>
<td>2.96M</td>
<td>2.92M</td>
<td>2.43M</td>
</tr>
<tr>
<td># constrained faces</td>
<td>2.65M</td>
<td>19.40k</td>
<td>408.77k</td>
<td>428.44k</td>
<td>429.10k</td>
<td>350.78k</td>
</tr>
<tr>
<td>construction time</td>
<td>209.30 s</td>
<td>0.69 s</td>
<td>24.09 s</td>
<td>26.17 s</td>
<td>27.09 s</td>
<td>17.94 s</td>
</tr>
<tr>
<td>render time</td>
<td>3.62 s</td>
<td>1.51 s</td>
<td>4.29 s</td>
<td>2.82 s</td>
<td>2.01 s</td>
<td>1.10 s</td>
</tr>
<tr>
<td>avg # tetrahedra/ray</td>
<td>158.18</td>
<td>115.12</td>
<td>183.13</td>
<td>142.74</td>
<td>126.48</td>
<td>86.82</td>
</tr>
<tr>
<td><strong>Quality Tetrahedralization</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># tetrahedra</td>
<td>11.71M</td>
<td>123.29k</td>
<td>1.84M</td>
<td>1.96M</td>
<td>1.97M</td>
<td>1.86M</td>
</tr>
<tr>
<td># faces</td>
<td>23.42M</td>
<td>246.90k</td>
<td>3.69M</td>
<td>3.92M</td>
<td>3.95M</td>
<td>3.73M</td>
</tr>
<tr>
<td># constrained faces</td>
<td>2.65M</td>
<td>21.26k</td>
<td>409.04k</td>
<td>430.69k</td>
<td>430.94k</td>
<td>357.32k</td>
</tr>
<tr>
<td>construction time</td>
<td>491.13 s</td>
<td>2.27 s</td>
<td>51.54 s</td>
<td>56.19 s</td>
<td>58.99 s</td>
<td>120.27 s</td>
</tr>
<tr>
<td>render time</td>
<td>1.01 s</td>
<td>0.42 s</td>
<td>0.75 s</td>
<td>0.63 s</td>
<td>0.48 s</td>
<td>0.61 s</td>
</tr>
<tr>
<td>avg # tetrahedra/ray</td>
<td>45.05</td>
<td>40.39</td>
<td>45.20</td>
<td>47.06</td>
<td>44.02</td>
<td>47.49</td>
</tr>
<tr>
<td><strong>kD-Tree</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># leaf nodes</td>
<td>4.01M</td>
<td>27.51k</td>
<td>992.63k</td>
<td>821.46k</td>
<td>848.34k</td>
<td>203.38k</td>
</tr>
<tr>
<td>construction time</td>
<td>42.30 s</td>
<td>0.30 s</td>
<td>10.62 s</td>
<td>10.72 s</td>
<td>10.00 s</td>
<td>2.48 s</td>
</tr>
<tr>
<td>triangles / leaf-node</td>
<td>2.56</td>
<td>2.54</td>
<td>2.42</td>
<td>2.49</td>
<td>2.48</td>
<td>2.31</td>
</tr>
<tr>
<td>render time</td>
<td>0.37 s</td>
<td>0.23 s</td>
<td>0.29 s</td>
<td>0.24 s</td>
<td>0.22 s</td>
<td>0.28 s</td>
</tr>
<tr>
<td>avg # nodes/ray</td>
<td>45.17</td>
<td>38.94</td>
<td>43.78</td>
<td>32.58</td>
<td>30.86</td>
<td>46.22</td>
</tr>
<tr>
<td>avg # intersections/ray</td>
<td>2.24</td>
<td>2.41</td>
<td>2.19</td>
<td>2.43</td>
<td>2.40</td>
<td>2.20</td>
</tr>
</tbody>
</table>